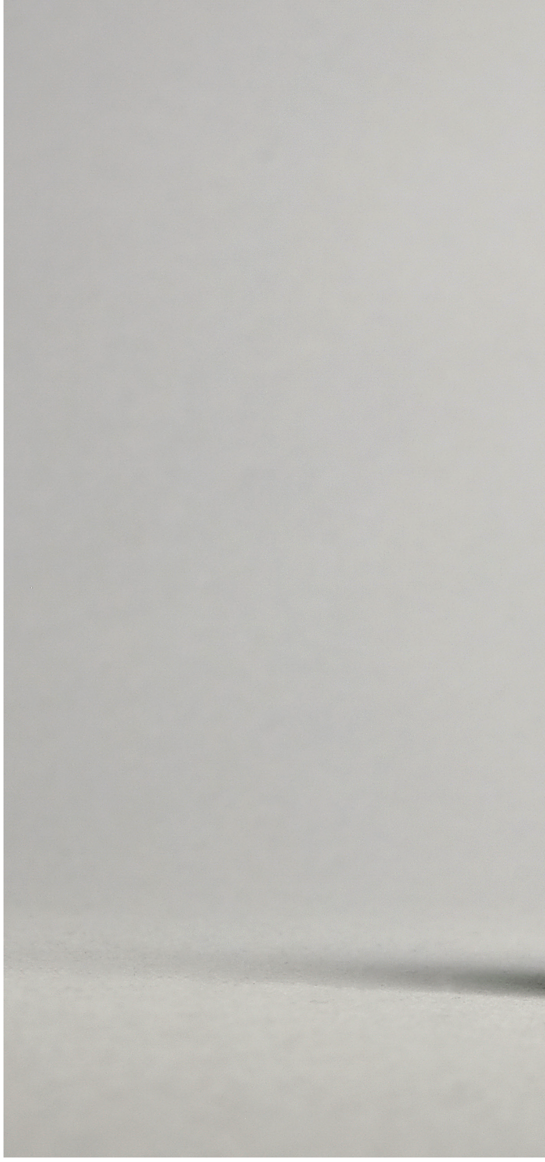
DOI:10.1145/3563901

Article development led by acmqueue
queue.acm.org

The return of a popular feature that shares the joy and utility of reading CS research between academics and their counterparts in industry.

BY MARTIN KLEPPMANN

Research for Practice: Convergence



IT IS WITH great pride and no small amount of excitement that I announce the reboot of Research for Practice. Beginning at its inception in 2016, Research for Practice brought both seminal and cutting-edge research—via careful curation by experts in academia—within easy reach for practitioners who are too busy building things to manage the deluge of scholarly publications. We believe the series succeeded in its stated goal of sharing "the joy and utility of reading computer science research" between academics and their counterparts in industry. We are delighted to rekindle the flame after a three-year hiatus.

For this first installment, we invited Martin Kleppmann, research fellow and affiliated lecturer at the University of Cambridge, to curate a selection of recent research papers in a perennially interesting domain: convergent or "eventual consistent" replicated systems. His expert analysis circles the topic, viewing it through the lens of recent work in four distinct research domains: systems, programming languages, human-computer interaction, and data management. Along the way, readers will be exposed to a variety of data structures, algorithms, proof techniques, and programming models (each described in terms of a distinct formalism), all of which attempt to make programming large-scale distributed systems easier. I hope you enjoy his column as much as I did.

—*Peter Alvaro*

Peter Alvaro is an associate professor of computer science at the University of California Santa Cruz, where he leads the Disorderly Labs research group (disorderlylabs.github.io).

In distributed systems, there are—broadly speaking—two approaches to data consistency: consensus or convergence. The consensus approach can be implemented with algorithms such as Paxos or Raft, and it ensures strong consistency, which means making the dis-

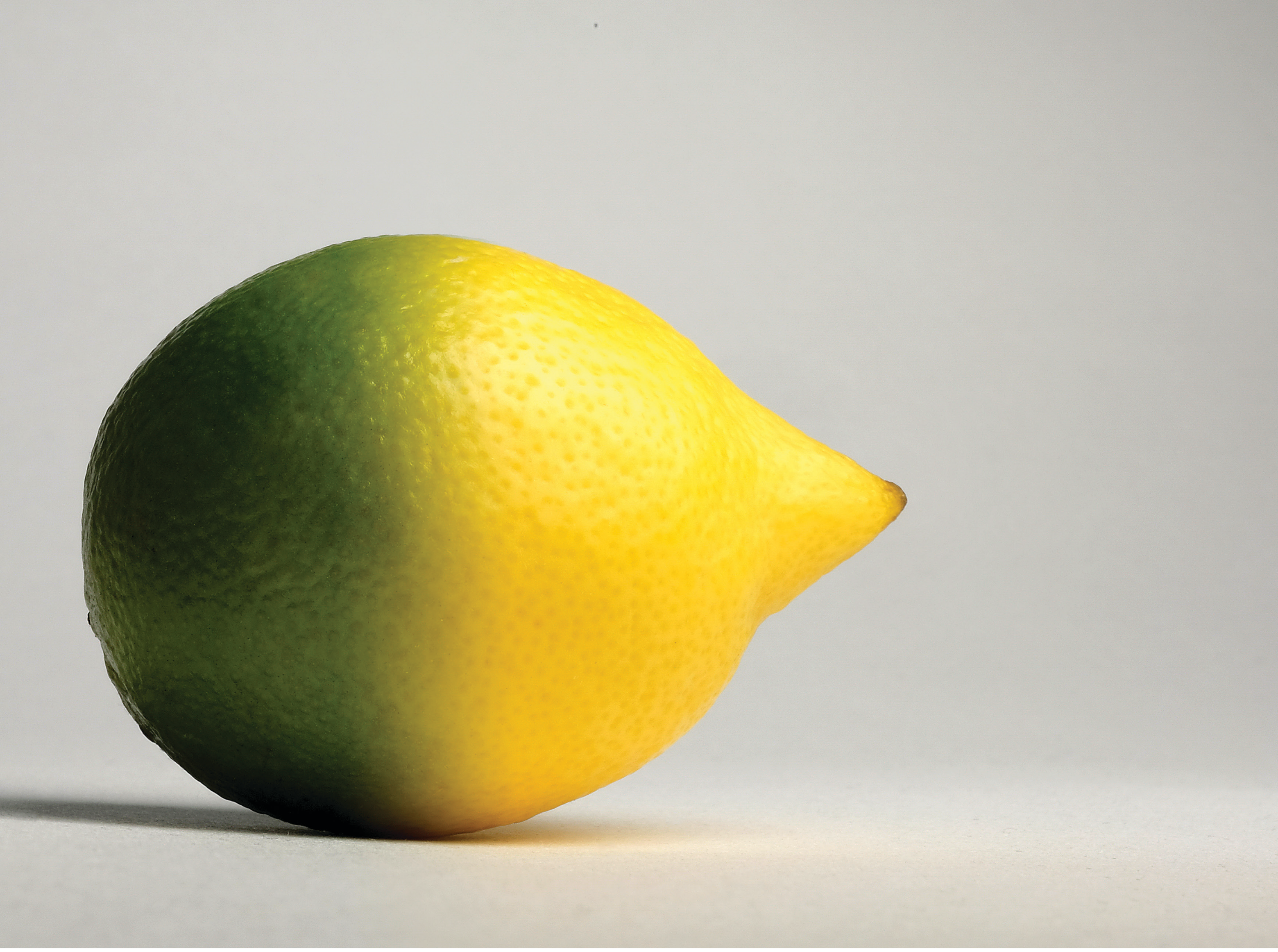


IMAGE BY BEN SLATER

tributed system appear as if it were not distributed (linearizable) and as if there were no concurrency (serializable). This approach makes the system easy to use, but it comes at the cost of performance, scalability, and the kinds of faults that can be tolerated, because every update needs to wait for a reply from other nodes before it can complete.

The alternative approach, convergence, is more commonly known as *eventual consistency*. In this model, different nodes are allowed to process updates independently, without waiting for each other. This is typically faster, more robust, and more scalable, but it leads to nodes having temporarily inconsistent versions of the data. As those nodes communicate with each other, those inconsistencies must be resolved—that is, they should *converge* toward the same state.

Convergence is such a useful idea that different research communities have developed several ways of achieving it. This article looks at four variations on the theme of convergence, drawn from four areas of computer science. I have selected five recent articles that provide introductions to each of the techniques for convergence.

Conflict-Free Replicated Data Types

N. Preguiça.
Conflict-free Replicated Data Types: An Overview (June 2018); https://arxiv.org/abs/1806.10254

A conflict-free replicated data type (CRDT) is a data structure that can be modified concurrently on several nodes and provides a built-in algorithm for merging those updates back together again. CRDTs have been created for a variety of data types, such as sets, lists, key-value maps, graphs, counters, and JSON (JavaScript Object Notation) trees. They are used in server-side databases such as Microsoft's Azure CosmosDB and Redis Enterprise, as well as client-side libraries for collaboration software such as Automerge and Yjs.

CRDTs ensure convergence through commutativity—that is, whenever two nodes have processed the same updates, they will be in the same state, even if the updates arrived in a different order. They achieve this property by adding some metadata to the actual data structure: For example, many algorithms associate a unique ID with each operation and use that ID later to refer to parts of the data structure. This makes the operations unambiguous when there are concurrent updates. By carefully managing this metadata, CRDTs ensure concurrent operations commute, enabling different replicas to merge their state and converge.

Operational Transformation

D. Sun, C. Sun, A. Ng, and W. Cai.
Real differences between OT and CRDT in correctness and complexity for consistency maintenance in co-editors. In *Proceedings of the ACM on Human-Computer Interaction 4*, CSCW1, article 21, (May 2020), 1–30; https://dl.acm.org/doi/10.1145/3392825

Operational transformation (OT) is

most used in real-time collaborative editors such as Google Docs, and it ensures whenever several users concurrently update the document, they converge to the same state. For plain text, the data structure is a linear sequence of characters that can be updated by inserting or deleting characters at any position. This idea has also been generalized to rich text, spreadsheets, and other file types. Such applications can be implemented with CRDTs as well, but many existing collaboration products use OT. The OT algorithm allows concurrent operations to be reordered through rules that are more restrictive than the general commutativity used by CRDTs.

OT is a much older technique than CRDTs; in fact, CRDTs were created in response to several flawed OT algorithms that were published in the 1990s and early 2000s. Today, both OT and CRDTs are widely used, and the trade-offs between them are nuanced. The suggested article is written by proponents of the OT approach, and its critique of CRDTs is unusually polemic for an academic paper. Even though I do not agree with everything the authors say, it's interesting to follow the spectacle of a heated debate.

Mergeable Replicated Data Types

G. Kaki, S. Priya, KC Sivaramakrishnan, and S. Jagannathan.
Mergeable replicated data types. In *Proceedings of the ACM Conference on Programming Languages 3*, OOSPLA, article 154 (Oct. 2019), 1–29; https://dl.acm.org/doi/10.1145/3360580

Mergeable replicated data types (MRDTs) is an alternative take on convergence that is based on ideas from version-control systems such as Git. In Git, if two users independently edit the same part of a file, a user must resolve the merge conflict manually, whereas CRDTs and OT automatically merge concurrent updates without requiring any user interaction. MRDTs combine CRDT/OT-like automatic merging with Git-like version control.

MRDTs are data structures like CRDTs. The difference is that while CRDTs provide a function for merging one node's state with another, MRDTs merge two branches of a version history by not only looking at the latest state on each branch, but also considering the most recent common ancestor state of the two branches (that is, the commit after which the two branches diverged). The MRDT can therefore see what has changed on each of the branches since the common ancestor, which allows it to maintain less metadata than a CRDT. Instead, it must maintain the commit history graph, which some CRDTs can avoid. MRDT algorithms exist for counters, queues, sets, maps, binary trees, and other data structures.

Consistency as Logical Monotonicity (CALM) and Invariant Confluence

J.M. Hellerstein and P. Alvaro.
Keeping CALM: When distributed consistency is easy. *Commun. ACM 63*, 9, (Sept. 2020), 72–81; https://dl.acm.org/doi/10.1145/3369736
P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J.M. Hellerstein, and I. Stoica.
Coordination avoidance in database systems. In *Proceedings of the VLDB Endowment 8*, 3 (2014), 185–196; http://www.vldb.org/pvldb/vol8/p185-bailis.pdf

The CRDT/OT/MRDT approaches are great in the situations for which they have been designed, but they are not sufficient to implement every type of software. In particular, if you need to manage some kind of limited resource—for example, to ensure that customers do not spend more money than they have in their accounts, or that you do not sell the same seat in a theater or airplane to more than one person, or that you do not promise the last in-stock item in the warehouse to more than one buyer—then you cannot just let each node update its state independently from other nodes. Some sort of coordination is required to decide which transaction goes through and who gets the seat or the last item in stock, because operations that consume the resource are mutually exclusive. This coordination could be implemented as a consensus algorithm or a locking protocol, for example.

The question then is: What general principle tells us when to use CRDTs and friends, and when stronger guarantees such as consensus are needed? The CALM theorem provides a precise answer to this question: Coordination can be avoided if the program is logically monotonic. CRDT/OT/MRDT algorithms are all ways of writing logically monotonic programs (the state of a data structure is determined by a monotonically growing set of updates); with such programs, the inputs can arrive in any order without affecting the output. On the other hand, managing access to a limited resource is a nonmonotonic operation and therefore requires coordination among the nodes in the system.

An alternative but related approach is to use the concept of *invariant confluence*. An invariant is confluent if two nodes can independently make updates that preserve the invariant, and you can be sure that the invariant continues to be satisfied when you merge the updates. Say you have an invariant such as "no seat in the theater is sold to more than one person." This example is not confluent because one node may sell an empty seat (which is valid), another node may independently sell the same seat (also valid), but the merge of the two updates violates the invariant. On the other hand, a referential integrity (foreign key) constraint is confluent if you only insert but don't delete. If all invariants are confluent, an application can be coordination-free, whereas nonconfluent invariants require coordination.

Conclusion

An interesting detail about these four approaches is that they have arisen from totally different areas of computer science: CRDTs come from the operating systems community, OT from human-computer interaction, MRDTs from programming languages, and CALM/invariant confluence from databases. Each community has applied its own style of thinking to the idea of convergence, which sometimes leads to misunderstandings of each other's work, especially as the different communities don't always talk to each other as much as you might hope. Taken together, however, this diverse set of perspectives gives us a stronger set of tools to apply to real-world problems. C

Martin Kleppmann is a research fellow at TU Munich, Germany, where he works on distributed systems security and collaborative software. Previously, he was a research fellow at the University of Cambridge, U.K. He has worked as a software engineer and entrepreneur at two startups and developed large-scale data infrastructure at LinkedIn. He is the author of *Designing Data-Intensive Applications* (O'Reilly Media).